Lab 7: Pointer Analysis

Synopsis

Writing a "division-by-zero" static analysis for C programs as an LLVM pass that handles pointer aliasing and dynamically allocated memory.

Objective

The goal of this lab is to extend the static **divide-by-zero** sanitizer in Lab 6 to perform its analysis in the presence of pointers. You will combine the dataflow analysis from the previous lab with a flow-insensitive pointer analysis, resulting in a more comprehensive overall static analysis.

Setup

The skeleton code for Lab 7 is located under `/lab7`. We will frequently refer to the top level directory for Lab 7 as `lab7` when describing file locations. This lab is built upon your work from Lab 6, so you can reuse most of your content from the `/lab6/src` directory.

Step 1.

The following commands set up the lab, using the CMake/Makefile pattern seen before.

```
/lab7$ mkdir build && cd build
/lab7$ cmake ..
/lab7$ make
```

Among the files generated, you should see `DivZeroPass.so` in the `build` directory, similar to the previous lab. In this lab you will modify `src/ChaoticIteration.cpp`, `DivZeroAnalysis.cpp`, and `Transfer.cpp`. Most of these changes can be copied over from the previous lab and then be modified to suit the new requirements.

We are now ready to run our bare-bones lab on a sample input C program.

Step 2.

Before running the pass, the LLVM IR code must be generated.

```
/lab7/test$ clang -emit-llvm -S -fno-discard-value-names -Xclang -disable-O0-optnone -c test03.c -o test03.ll
/lab7/test$ opt -load ../build/DivZeroPass.so -DivZero test03.ll
```

The first line (`clang`) generates LLVM IR code from the input C program `test03.c`. The next line (`opt`) runs your pass over the compiled LLVM IR code.

In prior lab, we used an intermediate step with the argument `-mem2reg` which promoted every AllocaInst to a register, allowing your analyzer to ignore handling pointers in this lab. However, in this lab we no longer do that so you will extend your previous code to handle pointers.

Upon successful completion of this lab, the output should be as follows:

```
/lab7/test$ opt -load ../build/DivZeroPass.so -DivZero test03.ll
Running DivZero on f
Potential Instructions by DivZero:
   %div = sdiv i32 1, %2
```

Format of Input Programs

The input format of this lab is the same as that of Lab 6 except now you will handle pointers:

- You *can* ignore precisely handling values other than integers but your LLVM pass must not raise a segmentation fault when encountered with other kinds of values.
- You *must* handle assignments, arithmetic operations (+, -, *, /), comparison operations (<, <=, >, >=, ==, !=), and branches.
- You *do not* have to handle XOR, OR, AND, and Shift operations precisely but your program must not raise a segmentation fault in these cases.
- Input programs *can* have if-statements and loops.
- User inputs are *only* introduced via the set of functions where the provided `isInput` function returns `True`.
- You *can ignore* other call instructions to other functions.

Lab Instructions

In this lab, you will extend the **divide-by-zero** analysis that you implemented in Lab 6 to analyze and catch potential **divide-by-zero** errors in the presence of aliased memory locations.

During lecture, you learned that introducing aliasing into a language makes reasoning about a program's behavior more difficult, and requires some form of pointer analysis. You will use a **flow-insensitive pointer analysis** — where we abstract away control flow and build a global **points-to graph** — to help your sanitizer analyze more meaningful programs.

Part 1: Function Arguments/Call Instructions

Step 1.

Recall that in previous lab, all of the test programs were basic functions that accepted no arguments.

For example:

```c
void f() {
    int x = 0;
    int y = 2;
    int z;
    if(x < 1) {
        z = y / x; // divide-by-zero within branch
    }
}
```

The function `f()` has no arguments in its signature. Realistically, functions can accept any number of variables, and even of different types (but for this lab, consider all arguments as `int`'s).

So in `doAnalysis`, you will need to handle functions with arguments and set up their domains accordingly.

Step 2.

Familiarize yourself with the `doAnalysis()` routine that acts as the entrypoint to your **divide-by-zero** LLVM pass. In last lab, you implemented the chaotic iteration algorithm here. For Lab 7, the function signature for `doAnalysis()` has now changed slightly to include a **PointerAnalysis** object. We will go over this in Part 2.

```cpp
/**
 * @brief This function implements the chaotic iteration algorithm using
 * flowIn(), transfer(), and flowOut().
 *
 * @param F The function to be analyzed.
 */
void DivZeroAnalysis::doAnalysis(Function &F, PointerAnalysis *PA)
```

Step 3.

Given an arbitrary function `F` passed into your `doAnalysis()` routine, find the arguments of the function call and instantiate abstract domain values for each argument. Note that the object `F` here is of type `Function`, which can be used to find all the arguments available.

Furthermore, once you've initialized these starting argument abstract values, pass these values into your existing implementation of the **divide-by-zero** pass such that these variables get propagated throughout the entire **reaching definitions analysis**.

Step 4.

In addition to handling arguments of the function `F` being analyzed, we also want to cover other function calls made within the program.

We've seen this before with this function:

```c
void main() {
    int x = getchar();
    int y = 5 / x;
    return 0;
}
```

In the above example, `getchar()` is an external function call made without arguments that returns an `int`. Update your analysis to handle arbitrary `CallInst` instructions, but only if the return type is an `int`.

Part 2: Store/Load Instructions

Step 1.

As mentioned above, there's a change made to the former doAnalysis() function:

```cpp
void DivZeroAnalysis::doAnalysis(Function &F, PointerAnalysis *PA)
```

In addition, we have modified the signature of the `transfer` function used in Lab 6:

```cpp
void DivZeroAnalysis::transfer(Instruction *I, const Memory *In, Memory *NOut,
                               PointerAnalysis *PA, SetVector<Value *> PointerSet)
```

Please make sure when reusing code from the previous assignment that you copy your implementation details and function contents, but **leave the function signatures intact!**.

These arguments are necessary as we explore pointer aliasing.

To help understand how the code is different from lab6 and how it's tied together, consider the following snippet from `DivZeroAnalysis::runOnFunction()`:


```cpp
bool DivZeroAnalysis::runOnFunction(Function &F) {
  outs() << "Running " << getAnalysisName() << " on " << F.getName() << "\n";

  // more code here...
  PointerAnalysis *PA = new PointerAnalysis(F);
  doAnalysis(F, PA);
  // more code here...
}
```

And the following snippet from `DivZeroAnalysis::doAnalysis()`:

```cpp
void DivZeroAnalysis::doAnalysis(Function &F, PointerAnalysis *PA) {
    for(inst_iterator I = inst_begin(F), E = inst_end(F); I != E, ++I) {
        WorkSet.insert(&(*I));
        PointerSet.insert(&(*I));
    }
    // more code here...
    transfer(I, In, NOut, PA, PointerSet);
    // more code here...
}
```

And, note that the transfer function now gets PointerAnalysis and a PointerSet as inputs. Keep this in mind when reusing your code from Lab 6.

Step 2.

At a high level, you will modify the `transfer()` function in `Transfer.cpp` to perform a more sophisticated **divide-by-zero** analysis by keeping track of pointers.

The code for `PointerAnalysis` is in `src/PointerAnalysis.cpp` and it includes the implementation of various methods needed to use the pointer aliasing. After the Pointer analysis is run of `F`, the `PointerAnalysis *PA` object will contain the result of the pointer analysis run on the function, and `PointerSet` will contain all pointers from the Function.

We will discuss in more detail what this `PointerAnalysis` class does in the following sections, but read through the docstrings and the code and make sense of what is being done in each of the methods provided.

Modeling LLVM alloca, store, and load.

Here we provide an interface for working with pointers in LLVM.

You may use this as is fallback, but you are also free free to model references in LLVM as you wish.

For this lab, we have disabled the `mem2reg` pass used in Lab 6. As such, LLVM will create a memory cell for every C variable. As a result you will not see any **phi-nodes**, and you will not necessarily need the code segments in which you implemented for handling them in Lab 6.

Consider the following code:

`int f() {`		
	`I1: %a = alloca i32, align 4`	
	`I2: %c = alloca i32*, align 4`	
	`I3: %x = alloca i32, align 4`	
`int a = 0;`	`I4: store i32 0, i32* %a, align 4`	`M[variable(I1)] = 0`
`int *c = &a;`	`I5: store i32* %a, i32** %c, align 8`	`M[variable(I2)] = variable(I1)`
	`I6: %0 = load i32*, i32** %c, align 8`	`M[variable(I6)] = M[variable(I2)]`
	`I7: %1 = load i32, i32* %0, align 4`	...
	`I8: %div = sdiv i32 1, %1`	...
`int x = 1 / *`	`I9: store i32 %div, i32* %x, align 4`	...
	`I10: %2 = load i32, i32* %x, align 4`	...
`return x;`	`I11: ret i32 %2`	...
`}`		...

As in Lab 6, the `variable()` method is still used to encode the variable of an instruction.

Building the Points-To Graph.

The `PointerAnalysis` class builds a points-to graph that you will use in your `transfer` function. `PointsToInfo` represents a mapping from variables to a `PointsToSet`, which represents the set of allocation sites a variable may point to.

To help model the memory location that corresponds to a variable `%a` (i.e., `variable(I1)`), we provide a function `address`, which you can use to encode the memory address (`address(I1)`) of a variable when building the `PointsToSet`.

Instruction `I2` will be similarly analyzed.

At `I5`, the memory location allocated at `I2` (i.e., `address(I2)`) will store the memory location allocated at `I1` (i.e., `address(I1)`).

Additionally, the field `PointsTo` represents the complete points-to graph that will be constructed.

The implementation for the `PointerAnalysis` constructor that will go through all the instructions for a given `Function F` and populate `PointsTo` has been provided to you as part of the skeleton code in this assignment.

Additionally, we have also provided an `alias()` method which returns true if two pointers may be aliases to one another.

Step 3.

Using the `PointerAnalysis` object, augment your `transfer()` function in `Transfer.cpp` to take into account pointer aliasing during its analysis. This should be done by adding code to handle `StoreInst` and `LoadInst` instructions in the `transfer` function.

LoadInst

We can rely on the existing variables defined within the `In` memory to know the abstract domain should be assigned for the new variable introduced by a load instruction.

For example, given a load instruction as follows:

```
%2 = load i32, i32* %1, align 4
```

This is loading the value of the pointer at `%1` into a new variable `%2` of type `i32`. So the abstract domain for `%2` should be the same as the abstract domain for `%1`.

With the addition of pointers, we can also have:

```
%1 = load i32*, i32** %d, align 8
```

This is loading the value of the pointer at `%d` (which itself is a pointer) into a new variable `%1` of type `i32*`.

Note the extra `*` characters in the load instruction's type (`load i32*`) compared to the previous example. You can retrieve this load instruction's type using `getType()`, and further check the type using methods like `isIntegerTy()` or `isPointerTy()`.

StoreInst

Store instruction can either add new variables or overwrite existing variables into our memory maps.

For example, given a store instruction as follows:

```
store i32, 0, i32* %a, align 4
```

This is storing the value of `0` into variable `%a`.

You should be familiar with retrieving these operands using `getOperand()`, but you can also use `getValueOperand()` and `getPointerOperand()` methods respectively. With the addition of pointers, we can also have:

```
store i32* %a, i32** %c, align 4
```

Now we're storing the pointer at `%a` into variable `%c`, which is a pointer to a pointer. We can again use type information from `getType()` on each of these operands to determine whether pointer-aliasing may apply.

This clearly complicates our abstract domain analysis - if some further instruction updates the value of `%a`, we not only need to update the abstract value of `%c`, but also consider updating the abstract value of other pointers that point to `%a`. This also applies to changes made to `%c` which is what happens in the `test03.c` example.

```c
int f() {
    int a = 1;
    int *c = &a;
    int *d = &a;
    *c = 0;
}
```

To resolve these cases, we can rely on the points-to graph constructed in `PointerAnalysis`.

We'll need to iterate through the provided `PointerSet`: if we come across some instance where there exists a may-alias (`PA->isAlias()` returns `true`), this essentially means there's an edge that connects the pointer values between two variables. Once we know what connections exist, we will need to get each abstract value, join them all together via `Domain::join()`, then proceed to update the current assignment as well as **all** may-aliased assignments with this abstract value. This ensures that all pointer references are in-sync and will converge upon a precise abstract value in our analysis.

Submission

Once you are done with the lab, you can create a `submission.zip` file by using the following command:

```
lab7$ make submit
...
submission.zip created successfully.
```

Then upload the submission file to Gradescope.